St. Louis Clojure

(not (eq clojure common-lisp))

Christopher Mark Gore

http://www.cgore.com

cgore@cgore.com

@cgore

Tuesday, June 17, AD 2014

Computer Science!

It's like math!

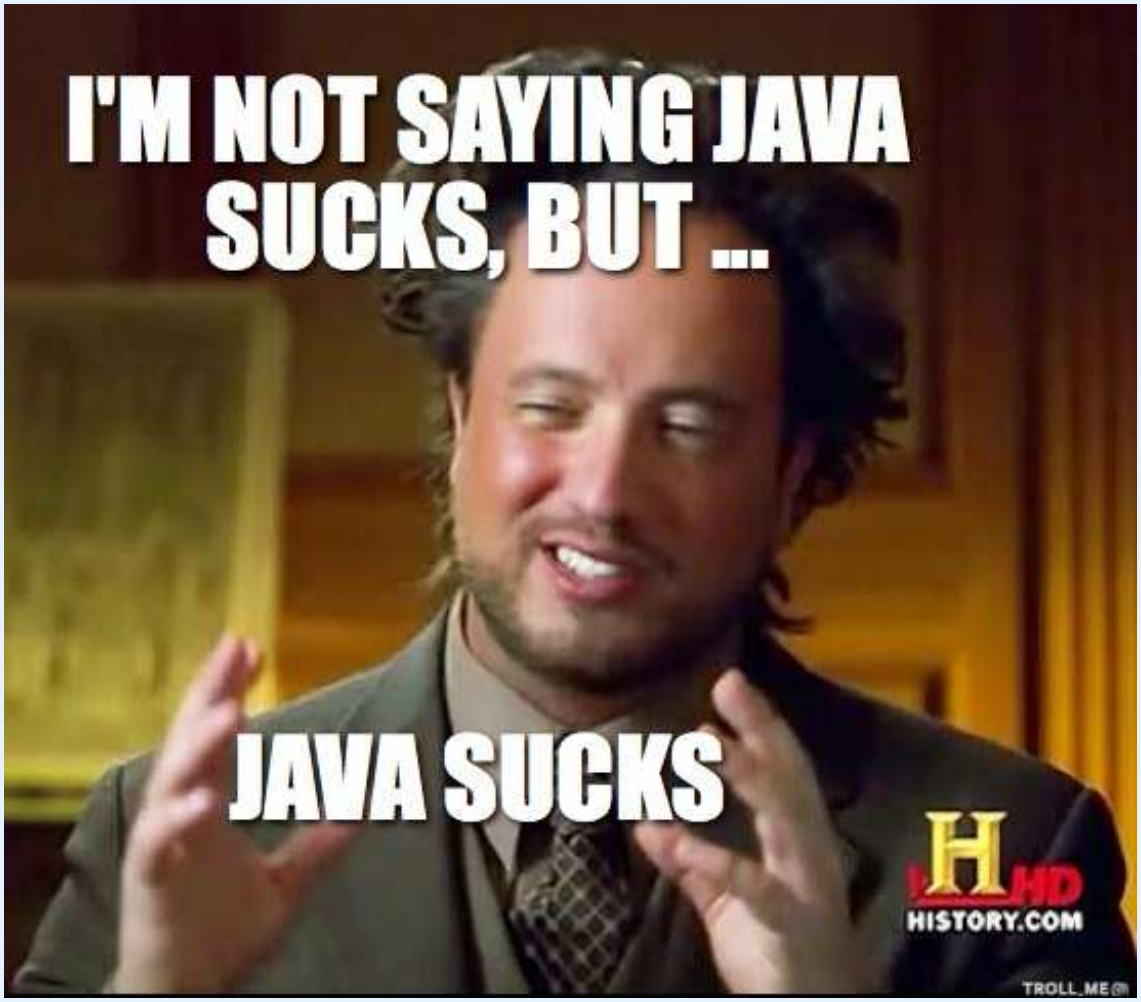
Axiom 1. *Lisp* = AWESOME!

Axiom 2. *Java* = CRAP!

Definition 1. *Clojure* = *Lisp* + *Java*.

Question 1. *Does* AWESOME! + CRAP! = AWESOME!,
or CRAP!,
or possibly just MEH*?*

Java is CRAP! . . .


I'M NOT SAYING JAVA SUCKS, BUT ...
JAVA SUCKS
HD
HISTORY.COM
TROLL_ME

. . . Java is CRAP! . . .

Here are some of the reasons why Java (the programming language) sucks.

- No lexically scoped local functions,
- No macro system,
- No inlined functions,
- Static methods aren't really class methods,
- No multiple inheritance (interfaces suck),
- Integers aren't objects,
- `String` **is** `final`,
- No `typedef`,
- Still has integer overflow,

. . . Java is CRAP!

- `BigInteger` exists, but isn't integrated at all,
- No real enums, and no easy way to make them,
- No pointers, not even a weak form of them,
- No list literals,
- CamelCaseIsStupidButEveryJavaProgramSeemsToUseIt,
- AndYesTheVariableNamesReallyAreThisLongToo,
- . . .

But Clojure isn't Java, so none of these are problems anymore!

Access to All of Java's Libraries is AWESOME!

As a giant conspiracy to make programmers less effective, Java was pushed all through the late 1990's and early 2000's in many major universities. Because of this, many, many good libraries exist for the JVM, because many, many programmers left school thinking that programming = Java.

- GUIs: SWING, SWT, . . .
- Parsing HTML, XML, JSON, . . .
- Databases: JDBC, Hibernate, . . .

No matter what it is, there's a Java library to do it. This is one of the things that really does hold Common Lisp back, a lack of good libraries for a lot of normal, everyday stuff.

Lisp_1 versus Lisp_2 . . .

Lisp_2 Original Lisp, Common Lisp, Emacs Lisp, . . .

Lisp_1 Scheme, Clojure, . . .

In a Lisp_2, there are separate namespaces associated for variables and for functions. This means the following code works in Common Lisp:

```
(let ((list '(1 2 3 4)))
  (defun listier (list)
    (list (list list)))
  (listier '(1 2 3 4))) ; Returns '(((1 2 3 4)))
```

In a Lisp_1, functions are in the variable namespace, so you will need to call the variable `the-list`, `lst`, `l`, etc.

. . . Lisp_1 versus Lisp_2

Referencing functions is a little more verbose in Lisp_2.

```
(mapcar #'sin '(0.1 0.2 0.3 0.4))
```

Versus just a variable in a Lisp_1 like Clojure.

```
(map sin [0.1 0.2 0.3 0.4])
```

It basically is, most of the time, just this:

Lisp_1 Less typing with FP approaches.

Lisp_2 Better names for arguments and variables when it really is *just any old list/vector/map/etc.* (All of these functions should actually probably be named `make-list`, etc.)

And so, if you really prefer FP, you generally also really prefer Lisp_1. Otherwise, a Lisp_2 is more convenient.

Lisp_5 versus Lisp_6? . . .

The names Lisp_1 and Lisp_2 are actually somewhat incorrect. In actuality, $\text{Lisp}_1 = \text{Lisp}_5$ and $\text{Lisp}_2 = \text{Lisp}_6$. This is because there are actually a lot more namespaces than just variables and functions.

1. Values (Lisp_1)
2. Functions (in a Lisp_2), but also:
3. Blocks,
4. Tags,
5. Type names, and
6. Declaration names.

... Lisp_5 versus Lisp_6? ...

Values are *"nouns."*

```
(let ((x 1) (y 2)) (+ x y)) ; x and y are values.
```

Functions are *"verbs."*

```
(defun f (x) (+ x 4077)) ; f is a function.
```

Blocks are named lexical exit points:

```
(let ((x (random 1.0))) ;  x ∈ [0.0, 1.0)∀x
  (block b
    (if (< x 0.5)
      (return-from b 0) ; 50% chance
      (if (< x 0.75)
        (return-from b 1) ; 25% chance
        (return-from b 2))))) ; remaining 25%
```

. . . Lisp_5 versus Lisp_6? . . .

Tags are just like FORTRAN and BASIC:

```
(prog (name)
  get-name
    (format t "What␣is␣your␣name?␣")
    (setf name (read-line))
    (when (zerop (length name)) (go goodbye))
    (format t "Hi,␣~A!~%" name)
    (go get-name)
  goodbye
    (format t "Goodbye!'~%"))
```

That's right, Common Lisp has GOTO!

It seemed like a good idea in 1958.

. . . Lisp_5 versus Lisp_6? . . .

Type names are separate, and this provides the same naming convenience.

```
(defun equidimensional (a)
  (or (< (array-rank a) 2)
      (apply #'= (array-dimensions a))))
(deftype square-matrix (&optional type size)
  `(and (array ,type (,size ,size))
        (satisfies equidimensional)))
(defun something-cool (square-matrix)
  (declare (type square-matrix square-matrix))
  (do (cool (linear (algebra (stuff square-matrix))))))
```

... Lisp_5 versus Lisp_6?

Declarations are special directives to the underlying Common Lisp system.

```
(defun nonsense (k x z)
  (foo z x) ; First call to foo
  (let ((j (foo k x)) ; Second call to foo
        (x (* k k)))
    (declare (inline foo) (special x z))
    (foo x j z))) ; Third call to foo
```

You can make your own, and they are their own namespace.

```
(declaim (declaration author target-language))
(declaim (target-language ada))
(declaim (author "Harry␣Tweeker"))
```

$\mathbf{Lisp_n}$ $\forall n \in \mathbb{Z}^+$?

We can make our own namespaces!

```
(defmacro defun2 ...)
(defmacro funcall2 ...)
(defmacro function2 ...)
;; A few more ...
(defun foo (x) (format t "Hi␣~A!~%" x))
(defun2 foo (x) (format t "Bye␣~A!~%" x))
(foo "Johnny") ; Hi Johnny!
(funcall2 'foo "Johnny") ; Bye Johnny!
```

The point is, when we do `(f x y z)`, It's just a shortcut. We are actually really doing:

```
(funcall (function f) x y z)
```

Lambda is Better Than Fn?
Fn is Better Than Lambda?

This is really a problem with our inferior keyboards.

$\lambda > f$

Defun is Better Than Defn?
Defn is Better than Defun?

This is probably a keyboard problem too.

I think I know how to fix this.

Common Lisp's Cond is CRAP!...

```
(cond ((< age 1)  "infant")
      ((< age 13) "child")
      ((< age 18) "teenager")
      ((< age 65) "adult")
      (t          "ancient"))
```

"Common Lisp's `cond` has too many parens, lets get rid of them!"

```
(cond (< age 1)  "infant"
      (< age 13) "child"
      (< age 18) "teenager"
      (< age 65) "adult"
      :else      "ancient")
```

. . . Common Lisp's Cond is ~~CRAP!~~ AWESOME!

Actually, no it isn't, having implicit `progn`'s are nice.

```
(cond ((blue? the-color)
       (format t "Blue is great!~%")
       :blue)
      ((green? the-color)
       (format t "Green, the color of money.~%")
       :green)
      ((red? the-color)
       (format t "Blood red sun ...~%")
       red)
      (t (format t "What?~%")))
```

Perhaps add a `condn` that works like Common Lisp's `cond`?

Hashmap Syntax is AWESOME!

Here's how to make a hashmap in Clojure:

```
{:name "John␣McCarthy"
 :born [1927 09 04]
 :died [2011 10 24]}
```

How do we do that in Common Lisp?

```
(let ((h (make-hash-table)))
  (setf (gethash :name h) "John␣McCarthy"
        (gethash :born h) '(1927 09 04)
        (gethash :died h) '(2011 10 24)))
```

Generally for that sort of stuff, you will use a structure or a class instead, so the syntax is really irritating.[a]

[a] If you have my Σ library then this becomes much more reasonable, thanks to the `populate-hash-table` function.

Lazy Evaluation is AWESOME!

Naïve Approach Just write the code like you can load *all the data*, and then be confused when bad things happen.

Streaming Approach Instead, write the code so that you only take one at a time, and then handle it that way, saving memory.

Batched Approach Determine a good amount to take at a time, and handle it in batches.

The way I understand it, Clojure's lazy evaluation actually does batching of some sort, which is AWESOME! if true. How do we tune this?

Vector Syntax is . . . MEH

- Honestly, the brackets just seem to get in the way.
- I really wish `defn` didn't use a vector for it's arguments.
- They interfere with my normal approach to closing mismatched parens from Common Lisp, which is to just spam the close-paren key until the editor highlights one as bad, and then backspace.[a]

[a] This is less of a problem now that I have finally figured out Paredit mode.

Threading Operators are AWESOME!

How to make a REPL? Or how to make a LPER? Sometimes the order really does matter.

```
(-> read eval print loop) ; versus ...
(loop (read (eval (print))))
```

I really like `->>` and `as->` too. I think there are a lot more of these that are missing though, and just haven't been thought about yet.

The Final Analysis

- Clojure is a really good language choice these days.
- A lot of stuff should be borrowed into Common Lisp from Clojure.
- Building on top of the JVM was actually a really good idea to help it get off the ground in the beginning.
- Clojure has some really nice features on it's own too, and some interesting approaches to some things.

I just wish Clojure was a little less insistent about things sometimes, about doing things *the Clojure way.* It seems opinionated in the same way as Python: yes, 99% of the time it is correct, I shouldn't do it that way, I should do it the way it wants, but there is that other 1%.

Questions?